Applying IC-Scheduling Theory to Familiar Classes of Computations

Gennaro Cordasco* Univ. of Salerno

Grzegorz Malewicz† Google, Inc.

Arnold L. Rosenberg‡ Univ. of Massachusetts

September 21, 2006

Abstract

Earlier work has developed the underpinnings of *IC-Scheduling Theory*, an algorithmic framework for scheduling computations having intertask dependencies for Internet-based computing (IC, for short). The goal of the schedules produced by the Theory is to render tasks eligible for execution at the maximum possible rate, with the dual aim of: (*a*) utilizing remote clients' computational resources well, by always having work to allocate to an available client; (*b*) lessening the likelihood of the "gridlock" that ensues when a computation stalls for lack of tasks that are eligible for execution. While motivated by real computations, the Theory has been developed with computations represented abstractly, via directed acyclic graphs (dags). The current paper reconnects the abstract theory with an eclectic variety of real computations and computational paradigms, by illustrating how to schedule these computations optimally.

1 Introduction

Earlier work [9, 21, 22, 23] has developed *IC-Scheduling Theory*, an algorithmic framework for studying the problem of scheduling computations that have intertask dependencies for

*Dip. di Informatica e Applicazioni, Univ. di Salerno, Baronissi (SA) 84081, ITALY, cordasco@dia.unisa.it

†Dept. of Engineering, Google Inc., Mountain View CA 94043, USA, malewicz@google.com

‡Dept. of Computer Science, Univ. of Massachusetts Amherst, Amherst, MA 01003, USA, rsnbrg@cs.umass.edu

the several modalities of Internet-based computing (IC, for short)—including Grid computing (cf. [5, 13, 12]), global computing (cf. [6]), and Web computing (cf. [17]). The goal is to craft schedules that maximize the rate at which tasks are rendered eligible for allocation to remote clients (hence for execution), with the dual aim of: (a) enhancing the effective utilization of remote clients, by always having work to allocate to an available client; (b) lessening the likelihood of the "gridlock" that can arise when a computation stalls pending computation of already-allocated tasks.

IC-Scheduling Theory idealizes the problem of scheduling computations having intertask dependencies for IC, via the assumption that tasks are executed in the order of their allocation. (This assumption idealizes the hope that monitoring clients' past behaviors and current capabilities, as prescribed in [5, 16, 24], can render the desired order likely, if not certain.) Building on the case studies of [22, 23], we have developed in [21, 9, 10] what we hope will be the underpinnings of an algorithmic theory of how to schedule computations having intertask dependencies for IC. Simulation experiments reported in [19, 15] bolster this hope by showing that, in a wide range of circumstances, the Theory's schedules outperform those produced by popular heuristics.

While motivated by real computations, IC-Scheduling Theory has been developed with computations represented abstractly, via *directed acyclic graphs* (*dags*, for short).[1] The current paper attempts to reconnect the Theory by describing an eclectic variety of significant familiar computations and computational paradigms, providing for each:

1. an analysis of its intertask dependency structure;
2. a sketched formal analysis of how this structure is scheduled optimally by the theory;
3. a discussion of how this structure can be rendered *multi-granular* by clustering tasks to adjust task granularities, at least over a wide range, while maintaining a desirable intertask dependency structure. Multi-granularity is quite important in IC, since it allows one both to tailor task granularity for remote clients that have widely varying computing powers and to diminish the volume of inter-client communication (which proceeds over the Internet).

We illustrate each paradigm with one or more significant applicative computations. The reader should keep in mind throughout that our overriding concern when discussing each illustrative computation is how the structure of the computation's intertask dependencies is accommodated by IC-Scheduling Theory. Other issues—even critical ones such as communication load, which may influence one's decision about the computation's suitability for IC—are *not* our primary concern here (although we plan to address them in future work).

[1] All technical terms are defined in Section 2.

This paper is part of our multi-pronged attempt to assess the impact of IC-Scheduling Theory on "real" IC. In [19], an extension of the scheduling algorithm of [21] is compared, via simulated computations on four "real" scientific dags, against the "FIFO" dag-scheduling heuristic used by the Condor system [7]. In [15], the (unextended) algorithm of [21] is compared, via simulated computations on many artificially generated dags, against three natural dag-scheduling heuristics, including "FIFO." All of the simulations suggest that IC-scheduling theory has a significant positive impact on the scheduling problem for IC, either matching or improving the performance of competing schedulers.

Roadmap. Section 2 presents the technical background for our study, including a review of the highlights of IC-scheduling theory, from [9, 21]. Section 3 discusses the important class of *expansion-reduction* computations, which, notably, includes computations derived using the divide-and-conquer paradigm. Section 4 discusses the class of wavefront computations encountered in applications as varied as finite-element computations and the arrays that arise in computer vision. The development builds on studies in [22] and [23]. Section 5, which also builds on [23], discusses *butterfly-structured* computations, which include the important family of *convolutions*. Section 6 expands on the theme of expansion-reduction computations, considering more complicated modes of "expansion," such as the important parallel-prefix paradigm. Finally, Section 7 focuses on the important matrix-multiply computation, which is a gateway to a variety of linear-algebraic computations. Section 8 projects future directions in the development of IC-scheduling theory.

Related work. Our study builds directly on our earlier work that develops the nascent IC-scheduling theory. The fundamental notions of IC-scheduling theory are introduced in [22, 23], which also characterize and specify optimal schedules for several uniform dags (some of which appear in this paper). The examples of the preceding sources are developed into the seeds of a scheduling theory in [9, 21], whose conceptual contributions we have just described; these sources also provide a rich repertoire of building blocks for the complex dags that the theory can handle. Major extensions to the theory are currently being developed in [10]. A companion to the preceding sources, [20]—which is motivated by the fact that many dags do not admit an optimal schedule in the sense of [21]—pursues an orthogonal regimen for scheduling dags for IC, in which a server allocates *batches* of tasks periodically, rather than allocating individual tasks as soon as they become eligible. Optimality is always possible within the batched framework, but achieving it may entail a prohibitively complex computation. As described earlier, our work also relates closely to [19, 15], which discuss the simulation studies alluded to, as well as (in the first of these sources) describing the PRIO tool. In less directly related work, [14] presents a probabilistic approach to the problem of executing tasks on unreliable clients. Finally, the impetus for our study derives from the many exciting systems- and/or application-oriented studies of IC, in sources such as [5, 6, 12, 13, 16, 17, 24].

2 The Rudiments of IC-Scheduling Theory

2.1 Computation-Dags

A *directed graph* $\mathcal{G}$ is given by a set of *nodes* $N_{\mathcal{G}}$ and a set of *arcs* (or, *directed edges*) $A_{\mathcal{G}}$, each of the form $(u \to v)$, where $u, v \in N_{\mathcal{G}}$. (The arc is *oriented* from u to v.) A *path* in $\mathcal{G}$ is a sequence of arcs that share adjacent endpoints, as in the following path from node u_1 to node u_n: $(u_1 \to u_2)$, $(u_2 \to u_3)$, $\ldots$, $(u_{n-1} \to u_n)$. A *dag* (*directed acyclic graph*) $\mathcal{G}$ is a directed graph that has no cycles—so that no path of the preceding form has $u_1 = u_n$. When a dag $\mathcal{G}$ is used to model a computation, i.e., is a *computation-dag:*

- each node $v \in N_{\mathcal{G}}$ represents a task in the computation;
- an arc $(u \to v) \in A_{\mathcal{G}}$ represents the dependence of task v on task u: v cannot be executed until u is.

For any arc $(u \to v) \in A_{\mathcal{G}}$, u is a *parent* of v, and v is a *child* of u in $\mathcal{G}$. The *indegree* (resp., *outdegree*) of node u is its number of parents (resp., children). A parentless node of $\mathcal{G}$ is a *source*, and a childless node is a *sink*. $\mathcal{G}$ is *connected* if, when one ignores the orientations of its arcs, there is a path connecting every pair of distinct nodes.

2.2 A Quality Model for Executing Dags on the Internet

When one executes a computation-dag[2] $\mathcal{G}$, a node $v \in N_{\mathcal{G}}$ is ELIGIBLE (for execution) only after all of v's parents have been executed; hence, every source of $\mathcal{G}$ is always ELIGIBLE. We do not allow recomputation of nodes, so a node loses its ELIGIBLE status once it is executed. In compensation, after a node $v \in N_{\mathcal{G}}$ has been executed, there may be new nodes that are rendered ELIGIBLE; this occurs when v is their last parent to be executed. A *schedule* for a dag $\mathcal{G}$ is a rule for selecting which ELIGIBLE node to execute at each step of an execution of $\mathcal{G}$. We measure the quality of an execution of $\mathcal{G}$ by the number of ELIGIBLE nodes after each node-execution—the more, the better. (Note that *we measure time in an event-driven manner*, as the number of nodes that have been executed to that point.) Our goal is to execute $\mathcal{G}$'s nodes in an order that maximizes the production rate of ELIGIBLE nodes *at every step of the computation.* Any schedule for $\mathcal{G}$ that achieves this demanding goal is said to be **IC-optimal**.

The significance of IC optimality stems from the following scenarios. (1) Schedules that produce ELIGIBLE nodes more quickly may reduce the chance of the "gridlock" that could occur when remote clients are slow—so that new tasks cannot be allocated pending the return of already allocated ones. (2) If the IC Server receives a batch of requests for tasks at (roughly) the same time, then having more ELIGIBLE tasks available allows the Server to satisfy more requests, thereby increasing "parallelism."

[2]For brevity, we henceforth refer to "dags," without the qualifier "computation."

2.3 Tools for Crafting IC-Optimal Schedules

We describe tools for generating IC-optimal schedules to use with our sample computations.

2.3.1 Composition-based scheduling tools

The priority relation $\triangleright$. For $i = 1, 2$, let the dag $\mathcal{G}_i$ have n_i nonsinks, and let it admit the IC-optimal schedule Σ_i. If the following inequalities hold:[3]

$$\begin{array}{l}(\forall x \in [0, n_1])\ (\forall y \in [0, n_2]) : \\ \quad E_{\Sigma_1}(x) + E_{\Sigma_2}(y) \ \leq\ E_{\Sigma_1}(\min\{n_1, x+y\}) + E_{\Sigma_2}(\max\{0, x+y-n_1\}),\end{array} \tag{2.1}$$

then $\mathcal{G}_1$ *has priority over* $\mathcal{G}_2$, denoted $\mathcal{G}_1 \triangleright \mathcal{G}_2$. Informally, one never decreases IC quality by executing a nonsink of $\mathcal{G}_1$ whenever possible.

Generating complex dags via composition. The operation of *composition* is defined inductively as follows.

- Start with a set $\mathcal{S}$ of base dags.
- To compose dags $\mathcal{G}_1, \mathcal{G}_2 \in \mathcal{S}$—which could be the same dag with nodes renamed to achieve disjointness—thereby obtaining a composite dag $\mathcal{G}$:
  - Let $\mathcal{G}$ begin as the sum $\mathcal{G}_1 + \mathcal{G}_2$,[4] with nodes renamed if necessary to ensure that $N_{\mathcal{G}} \cap (N_{\mathcal{G}_1} \cup N_{\mathcal{G}_2}) = \emptyset$.
  - Select some set S_1 of sinks from the copy of $\mathcal{G}_1$ in the sum $\mathcal{G}_1 + \mathcal{G}_2$, and an equal-size set S_2 of sources from the copy of $\mathcal{G}_2$ in the sum.
  - Pairwise identify (i.e., merge) the nodes in the sets S_1 and S_2 (in an arbitrary way). The resulting node-set is $N_{\mathcal{G}}$; the induced set of arcs is $A_{\mathcal{G}}$.[5]
- Add the dag $\mathcal{G}$ thus obtained to the base set $\mathcal{S}$.

We denote the composition operation by $\Uparrow$ and say that $\mathcal{G}$ is *composite of type* $[\mathcal{G}_1 \Uparrow \mathcal{G}_2]$.

The dag $\mathcal{G}$ is a $\triangleright$*-linear composition* of the dags $\mathcal{G}_1, \ldots, \mathcal{G}_n$ if: (a) $\mathcal{G}$ is composite of type $\mathcal{G}_1 \Uparrow \cdots \Uparrow \mathcal{G}_n$; (b) $\mathcal{G}_i \triangleright \mathcal{G}_{i+1}$, for all $i \in [1, n-1]$.

The following results underlies the main scheduling algorithm of [21].

[3] $[a, b]$ denotes the set of integers $\{a, a+1, \ldots, b\}$.

[4] If $\mathcal{G}_1$ and $\mathcal{G}_2$ are *disjoint*, in the sense that $N_{\mathcal{G}_1} \cap N_{\mathcal{G}_2} = \emptyset$, then their *sum*, denoted $\mathcal{G}_1 + \mathcal{G}_2$, is the dag whose node-set is $N_{\mathcal{G}_1} \cup N_{\mathcal{G}_2}$ and whose arc-set is $A_{\mathcal{G}_1} \cup A_{\mathcal{G}_2}$.

[5] An arc $(u \to v)$ is *induced* if $\{u, v\} \subseteq N_{\mathcal{G}}$.

Theorem 2.1 ([21]). *Let $\mathcal{G}$ be a $\triangleright$-linear composition of $\mathcal{G}_1, \ldots, \mathcal{G}_n$, where each $\mathcal{G}_i$ admits an IC-optimal schedule Σ_i. The schedule Σ for $\mathcal{G}$ that proceeds as follows is IC optimal.*

1. *For $i = 1, \ldots, n$, in turn, Σ executes the nodes of $\mathcal{G}$ that correspond to nonsinks of $\mathcal{G}_i$, in the order mandated by Σ_i.*
2. *Σ finally executes all sinks of $\mathcal{G}$ in any order.*

2.3.2 Duality-based scheduling tools

The *dual* of a dag $\mathcal{G}$ is the dag $\widetilde{\mathcal{G}}$ that is obtained by reversing all of $\mathcal{G}$'s arcs (thereby interchanging sources and sinks). One can infer both IC-optimal schedules and $\triangleright$-priorities for a dag $\mathcal{G}$ from corresponding entities for $\widetilde{\mathcal{G}}$ (Theorems 2.2 and 2.3, respectively).

Scheduling-based duality. Let $\mathcal{G}$ be a dag with n nonsinks, $U = \{u_1, \ldots, u_n\}$, and N nonsources, $V = \{v_1, \ldots, v_N\}$. Let Σ be a schedule for $\mathcal{G}$ that executes U's nodes in the order $u_{k_1}, u_{k_2}, \ldots, u_{k_n}$ (followed by all of $\mathcal{G}$'s sinks). Each execution of a nonsink, say u_{k_j}, renders ELIGIBLE a (possibly empty) "packet" of nonsources, $P_j = \{v_{j,1}, \ldots, v_{j,i_j}\}$, of $\mathcal{G}$. Thus, Σ renders $\mathcal{G}$'s nonsources ELIGIBLE in a sequence of such "packets:"

$$P_1 = \{v_{1,1}, \ldots, v_{1,i_1}\},\ P_2 = \{v_{2,1}, \ldots, v_{2,i_2}\},\ \ldots,\ P_n = \{v_{n,1}, \ldots, v_{n,i_n}\}.$$

A schedule $\widetilde{\Sigma}$ for $\widetilde{\mathcal{G}}$ is *dual to* Σ if it executes $\widetilde{\mathcal{G}}$'s nonsinks—i.e., V's nodes—in an order of the form[6]

$$[[v_{n,1}, \ldots, v_{n,i_n}]],\ [[v_{n-1,1}, \ldots, v_{n-1,i_{n-1}}]],\ \ldots,\ [[v_{1,1}, \ldots, v_{1,i_1}]],$$

after which, $\widetilde{\Sigma}$ executes $\widetilde{\mathcal{G}}$'s sinks. ($\widetilde{\mathcal{G}}$ generally admits many schedules that are dual to Σ.)

Theorem 2.2 ([9]). *Let the dag $\mathcal{G}$ admit the IC-optimal schedule $\Sigma_{\mathcal{G}}$. Any schedule for $\widetilde{\mathcal{G}}$ that is dual to $\Sigma_{\mathcal{G}}$ is IC optimal.*

Priority-based duality. We can infer $\triangleright$-priorities via duality.

Theorem 2.3 ([9]). *For all dags $\mathcal{G}_1$ and $\mathcal{G}_2$: $\mathcal{G}_1 \triangleright \mathcal{G}_2$ if, and only if, $\widetilde{\mathcal{G}}_2 \triangleright \widetilde{\mathcal{G}}_1$.*

3 Alternating Expansion-Reduction Computations

3.1 The Abstract Dags

The structure of the dags. The computations we exemplify in this section are built via iterated composition from the two basic building blocks depicted in Fig. 1: the *Vee dag* $\mathcal{V}$ on the left and the *Lambda dag* Λ on the right. (Both are named for the shapes of

[6] $[[a, b, \ldots, c]]$ denotes a fixed, but unspecified, permutation of the set $\{a, b, \ldots, c\}$.

Figure 1: *The Vee dag* $\mathcal{V}$ (left) *and the Lambda dag* Λ (right).

their drawings. Note that Λ and $\mathcal{V}$ are dual to one another.) Via iterated composition: $\mathcal{V}$ is a typical building block for an "expansive" computation, in which one generates an out-tree to generate subcomputations—as, e.g., in the "divide" phase of a divide-and-conquer computation; Λ is a typical building block for a "reductive" computation, in which one generates an in-tree that accumulates previously computed results—as, e.g., in the recombination phase of a divide-and-conquer computation.

Our interest here is in multiphase computations, which compose alternating "expansive" computations—represented by out-trees—and "reductive" computations—represented by complete in-trees.[7] One such computation, which performs an expansive computation followed by a reductive one, is illustrated in Fig. 2, where the composition is represented explicitly. The out-tree at the left-bottom of the figure generates values; the in-tree at the

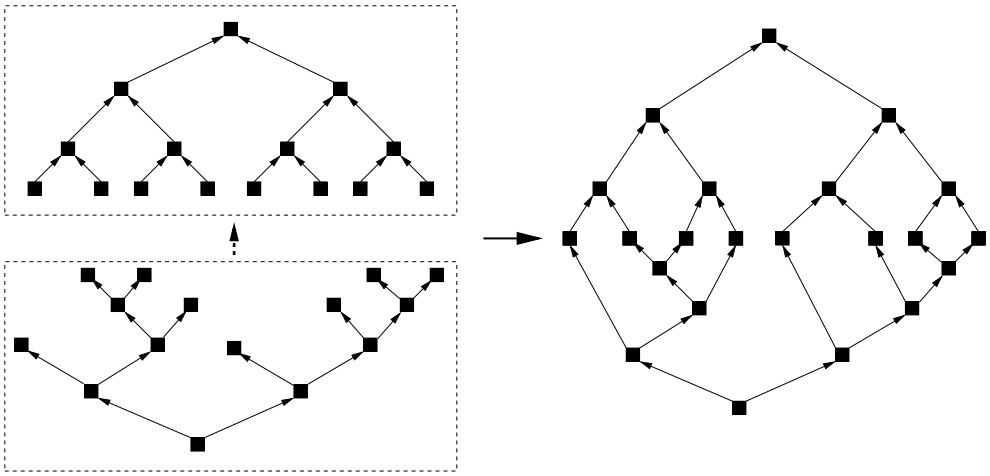

Figure 2: *A sample expansion-reduction computation.*

left-top accumulates the values. The two trees are composed into the *diamond* dag at the right by merging (in this case, all) sinks of the out-tree with sources of the in-tree.

Rendering expansion-reduction computations multi-granular. The out-tree in an expansion-reduction computation generates the values that the in-tree subsequently accu-

[7] Our restriction to *binary* trees in illustrations is irrelevant to the theory: any fixed degree works. Indeed, one finds degree-d analogues of both $\mathcal{V}$ and Λ in [21].

mulates. One can easily coarsen the tasks in such a computation by selectively truncating branches of the out-tree, together with mated portions of the in-tree, in a manner that leaves more of the overall computation to remote clients. We illustrate this process in Fig. 3, where we transform the diamond dag of Fig. 2 by coarsening two tasks. To sim-

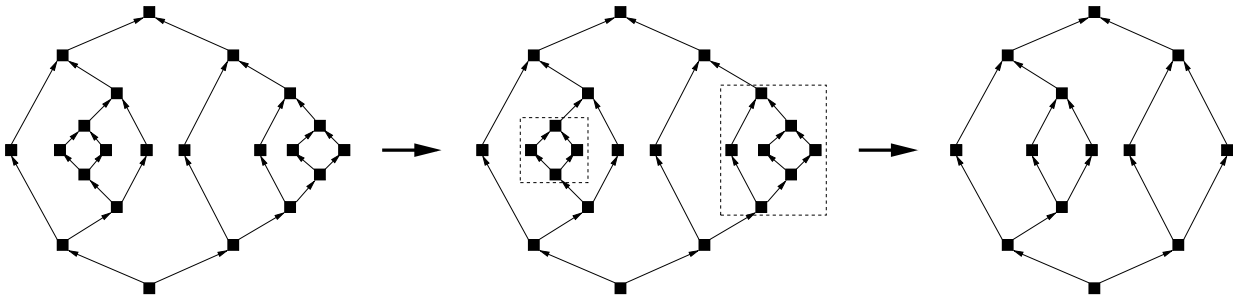

Figure 3: *Coarsening tasks in the dag of Fig. 2.*

plify our illustration, we replace the in-tree of Fig. 2 by the dual, $\widetilde{\mathcal{T}}$, of the out-tree $\mathcal{T}$; this is clearly just a simplification, not a required change. The reader should be able to extrapolate from this example to render other diamond dags multi-granular.

IC-optimal schedules for diamond dags. Every dag that represents an alternating expansive-reductive computation admits an IC-optimal schedule. We approach the task of deriving such a schedule in steps, beginning with out-trees and in-trees and progressing to increasingly complex compositions thereof.

Out-trees and in-trees. Note first that every out-tree is an iterated composition of the Vee dag $\mathcal{V}$, i.e., is composite of type $\mathcal{V} \Uparrow \cdots \Uparrow \mathcal{V}$. A trivial computation using (2.1) shows that $\mathcal{V} \rhd \mathcal{V}$, which verifies that every out-tree is a $\rhd$-linear composition, hence admits an IC-optimal schedule (Theorem 2.1). Indeed, easily, every schedule for an out-tree is IC optimal! Since every in-tree is dual to an out-tree, we infer from Theorem 2.2 that every in-tree admits an IC-optimal schedule.[8] Indeed:

> *A schedule for an in-tree $\mathcal{T}$ is IC optimal if, and only if, it executes the two sources of each copy of Λ in $\mathcal{T}$ in consecutive steps.* [23]

Diamond dags. Every diamond dag $\mathcal{D}$ is (by definition) a composition of an out-tree $\mathcal{T}$ and an in-tree $\mathcal{T}'$, hence, is composite of type $\mathcal{T} \Uparrow \mathcal{T}'$. Invoking the associativity of dag-composition [21], we infer that $\mathcal{D}$ is composite of some type $\mathcal{V} \Uparrow \cdots \Uparrow \mathcal{V} \Uparrow \Lambda \Uparrow \cdots \Uparrow \Lambda$, where the only uncertainty is the number of $\mathcal{V}$'s (which must match the number of Λ's). Since a trivial computation involving (2.1) shows that $\mathcal{V} \rhd \Lambda$, we see that every diamond dag is a $\rhd$-linear composition, hence admits an IC-optimal schedule (Theorem 2.1). Indeed

[8] Instead of invoking the Theorem, we could alter our discussion of out-trees by uniformly replacing "out-tree" by "in-tree" and "$\mathcal{V}$" by "Λ."

any schedule that executes all $\mathcal{T}$ using an IC-optimal schedule, then executes all of $\mathcal{T}'$ using an IC-optimal schedule, is IC optimal for $\mathcal{D}$.

More complicated alternations. Although our primary focus has been on diamond dags as exemplars of expansion-reduction computations, our analysis of diamond dags applies almost verbatim to a far broader family of alternating in-trees and out-trees, such as are exemplified in Fig. 4. Note from the rightmost dag in the figure that the numbers of leaves of composed out-trees and in-trees need not match.

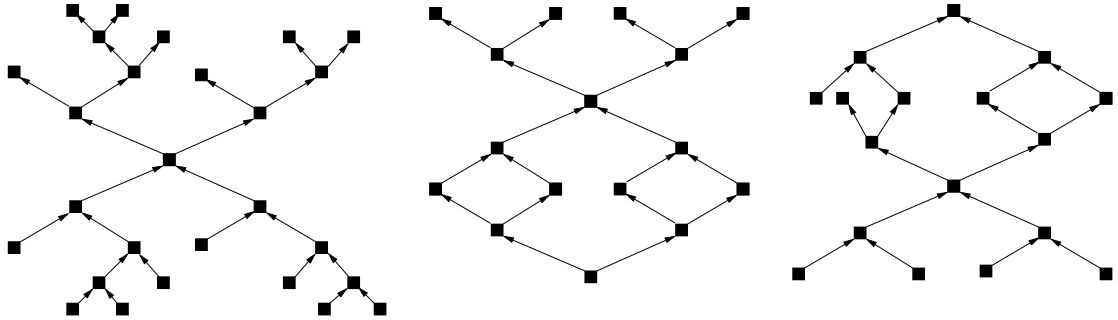

Figure 4: *Sample alternating expansion-reduction computations.*

To analyze these extended dags, we note that, although $\mathcal{T} \rhd \mathcal{T}'$ for any out-tree $\mathcal{T}$ and in-tree $\mathcal{T}'$, the converse does not hold. Nonetheless, the dag $\mathcal{G}$ of type $\mathcal{T}' \Uparrow \mathcal{T}$ that "merges" $\mathcal{T}'$'s single sink with $\mathcal{T}$'s single source, as in the leftmost dag in Fig. 4, admits an IC optimal schedule, because $\mathcal{G}$'s topology forces *every* schedule to execute all of $\mathcal{T}'$ before any of $\mathcal{T}$; hence, we need worry only about how to compute $\mathcal{T}$ and $\mathcal{T}'$ individually.

The preceding reasoning actually shows that any alternating composition of out-trees and in-trees of the composition-types depicted in Table 1 admits an IC-optimal schedule. (The superscript "(out)" identifies an out-tree; "(in)" identifies an in-tree.)

Tree-dag notation	Diamond-dag notation
$(\mathcal{T}_0^{(\text{out})} \Uparrow \mathcal{T}_0^{(\text{in})}) \Uparrow (\mathcal{T}_1^{(\text{out})} \Uparrow \mathcal{T}_1^{(\text{in})}) \Uparrow \cdots \Uparrow (\mathcal{T}_n^{(\text{out})} \Uparrow \mathcal{T}_n^{(\text{in})})$	$\mathcal{D}_0 \Uparrow \mathcal{D}_1 \Uparrow \cdots \Uparrow \mathcal{D}_n$
$\mathcal{T}_0^{(\text{in})} \Uparrow (\mathcal{T}_1^{(\text{out})} \Uparrow \mathcal{T}_1^{(\text{in})}) \Uparrow \cdots \Uparrow (\mathcal{T}_n^{(\text{out})} \Uparrow \mathcal{T}_n^{(\text{in})})$	$\mathcal{T}_0^{(\text{in})} \Uparrow \mathcal{D}_1 \Uparrow \cdots \Uparrow \mathcal{D}_n$
$(\mathcal{T}_1^{(\text{out})} \Uparrow \mathcal{T}_1^{(\text{in})}) \Uparrow \cdots \Uparrow (\mathcal{T}_n^{(\text{out})} \Uparrow \mathcal{T}_n^{(\text{in})}) \Uparrow \mathcal{T}_0^{(\text{out})}$	$\mathcal{D}_1 \Uparrow \cdots \Uparrow \mathcal{D}_n \Uparrow \mathcal{T}_0^{(\text{out})}$

Table 1: Diamond dags that admit IC-optimal schedules.

3.2 A Sample Computation: Numerical Integration

Alternating expansive-reductive dags arise in a number of important divide-and-conquer computations; we describe just one significant one.

A number of popular numerical integration algorithms proceed in the following way. (One can imagine the following as specifying the task that is computed in each node of the out-tree that represents the expansive portion of the computation.) Say that one wants to integrate a function F over an interval $[a_0 \leftrightarrow b_0]$.[9] One chooses a computationally simple functional form that provides a numerically adequate approximation to the area under F, at least over a very small interval. The Trapezoid Rule, e.g., uses a linear approximation to the area under F, while Simpson's Rule uses a quadratic approximation; for simplicity, we describe the Trapezoid Rule, which is based on the approximation

$$A(X,Y) \stackrel{\text{def}}{=} \frac{1}{2}(F(X)+F(Y))(Y-X).$$

One then computes two quantities

$$\begin{aligned} A_0 &= A(a_0, b_0) \\ A_1 &= A\left(a_0,\ \frac{1}{2}(a_0+b_0)\right) + A\left(\frac{1}{2}(a_0+b_0),\ b_0\right). \end{aligned}$$

A_0 is a linear approximation to the area under F over the interval $[a_0 \leftrightarrow b_0]$, while A_1 is the approximation obtained by splitting the interval $[a_0 \leftrightarrow b_0]$ in two, thereby making some accommodation for F's curvature within the interval. If the difference $|A_0 - A_1|$ is sufficiently small (relative to a predetermined tolerance), then the approximation A_0 is accepted, and the current task-node becomes a leaf of the out-tree. If the difference is too large—i.e., exceeds the tolerance—then the current task spawns two new tasks, representing the two summands of A_1, which become its children in the out-tree: the left child-task seeks to integrate F over the interval $[a_0 \leftrightarrow \frac{1}{2}(a_0+b_0)]$, the right child over the interval $[\frac{1}{2}(a_0+b_0) \leftrightarrow b_0]$. In terms of the depiction of $\mathcal{V}$ in Fig. 1, the variables w, x_0, x_1 represent the intervals over which the task-node must integrate the function F:

$$\begin{array}{lcl} \textbf{if} & w & = [a_0 \leftrightarrow b_0] \\ \textbf{then} & x_0 & = [a_0 \leftrightarrow \frac{1}{2}(a_0+b_0)] \\ \textbf{and} & x_1 & = [\frac{1}{2}(a_0+b_0) \leftrightarrow b_0] \end{array}$$

The initial task—the root of the out-tree—represents the entire interval $[a_0, b_0]$.

> **Note**. The preceding if-then-else prescription specifies intertask dependencies within the out-tree portion of the diamond dag. It does *not* specify a computation that *we* are doing.

The integration procedure ends by composing the final out-tree $\mathcal{T}$, whose leaves contain the area of F over the subintervals wherein a linear approximation to F suffices, with its

[9] We use the notation $[X \leftrightarrow Y]$ to denote the closed real interval $\{Z \mid X \le Z \le Y\}$.

dual in-tree $\widetilde{\mathcal{T}}$, which accumulates these areas; hence, the sink of $\widetilde{\mathcal{T}}$ provides the sought approximation to the area under F over the entire interval. In terms of the depiction of Λ in Fig. 1, the variables y_0, y_1, z represent the areas under F over the various subintervals:

$$\begin{array}{lrcl} \textbf{if} & y_0 & = & A(a_0,\ \frac{1}{2}(a_0 + b_0)) \\ \textbf{and} & y_1 & = & A(\frac{1}{2}(a_0 + b_0),\ b_0) \\ \textbf{then} & z & = & y_0 + y_1 \end{array}$$

The described computation thus generates a (possibly quite irregular) binary out-tree whose leaves contain the areas under the curve in regions that are small enough for a linear (Trapezoid Rule) or quadratic (Simpson's Rule) approximation to provide an adequate approximation to the true area. It then uses an in-tree to accumulate these areas over subintervals into the area of F over the entire interval $[a_0 \leftrightarrow b_0]$. By appropriately coarsening the diamond dag that represents this computation, one can decrease the volume of internode communication, as well as render the computation's tasks more coarse-grain.

4 Wavefront-Related Computations

This section builds on and extends the study of mesh-like computations in [22, 23].

4.1 The Abstract Dags

The structure of the dags. The computations we exemplify in this section all have the structures of two-dimensional meshes that are truncated along their diagnals; see Fig. 5. While the *out-mesh* on the lefthand side of the figure represents a large family of wavefront-

Figure 5: *An out-mesh (left) and an in-mesh (right).*

structured computations, it is useful also to consider the in-mesh on the righthand side. These latter dags (which are called *pyramid dags* in [8]) are useful when discussing multigranularity in mesh-like dags. Moreover, our analyses of in-meshes follow directly from our analyses of out-meshes, via duality (see Section 2.3.2).

IC-optimal schedules for mesh-like dags. Ad hoc arguments in [22, 23] show, respectively, that both out- and in-meshes admit IC-optimal schedules. A more interesting proof

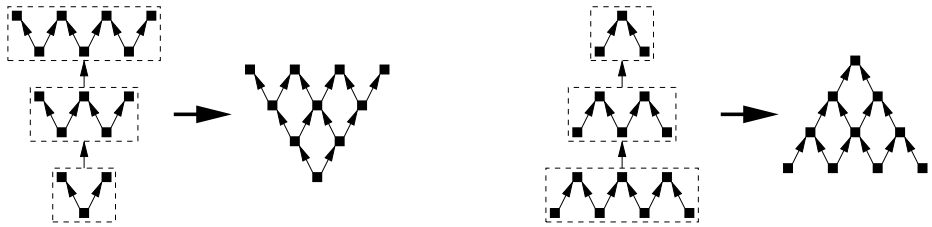

Figure 6: *The out-mesh and in-mesh as compositions.*

emerges by extrapolating from Fig. 6, which shows that every out-mesh is a composition of *W-dags*[10] having increasing numbers of sources. Since the schedule that executes a W-dag's sources consecutively is IC optimal and since smaller W-dags have $\triangleright$-priority over larger ones (both results from [21]), every out-mesh is a $\triangleright$-linear composition, hence admits an IC-optimal schedule. By duality, the same is true for in-meshes.

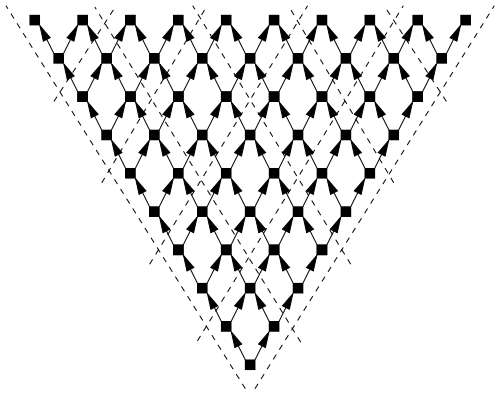

Figure 7: *Rendering an out-mesh multi-granular.*

Rendering wavefront computations multi-granular. There are many clustering strategies that allow one to coarsen the tasks in a mesh-like computation, but all are more complicated than their analogues for the other dags we study here, due to the "tighter" interdependencies in mesh-like dags. Fig. 7 suggests one scheme for coarsening an out-mesh's tasks by a factor of 4; this factor can be adjusted by sliding the dashed lines to generate equilateral "rectangles" and "triangles" whose "areas" determine the coarsening factor.[11] When all tasks have equal granularity, the coarsened mesh is just a smaller version of the original, fine-grained one, hence also admits an IC-optimal schedule. However, when tasks differ in granularities, the coarsened mesh loses the regularity of its fine-grained ancestor and may not admit an IC-optimal schedule. What is true of all such coarsenings, though,

[10] W-dags and M-dags are named for the Latin letters suggested by their topologies.

[11] Each "triangle" is an out-mesh, and each "rectangle" is the composition of an out-mesh and an in-mesh; hence both types of dags admit IC-optimal schedules.

is the important fact that the amount of computation represented by a coarsened task grows quadratically with the task's "sidelength," while the communication—a much dearer resource in IC—grows only linearly with "sidelength."

5 Butterfly-Structured Computations

5.1 The Abstract Dags

This section builds on and extends the study of the FFT computation in [23].

The structure of the dags. The computations we exemplify in this section are all built via iterated composition from the *butterfly building block* $\mathcal{B}$ of Fig. 8, so named for its shape in the drawing. A large variety of transformations effected by butterfly building blocks—

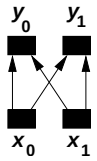

Figure 8: *The butterfly building block* $\mathcal{B}$.

i.e., specifications of y_0 and y_1 as functions of x_0 and x_1—lead to useful computations.

Notable among the dags constructed from butterfly building blocks is the *d-dimensional butterfly network* $\mathcal{B}_d$, for $d = 1, 2, \ldots$. The 1-dimensional butterfly network is the butterfly building block: $\mathcal{B}_1 = \mathcal{B}$. The 2- and 3-dimensional networks, $\mathcal{B}_2$ and $\mathcal{B}_3$, are depicted in Fig. 9. (The reader should easily be able to extrapolate to higher dimensional networks.)

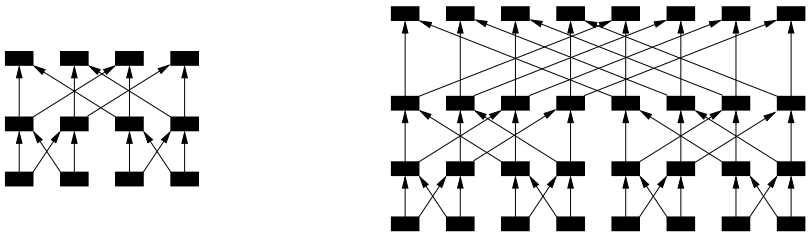

Figure 9: *The* 2*-dimensional* (left) *and* 3*-dimensional* (right) *butterfly networks.*

IC-optimal schedules for butterfly networks. The easiest way to begin deriving an IC-optimal schedule for the d-dimensional butterfly network $\mathcal{B}_d$ is to note (the well-known fact) that $\mathcal{B}_d$ is an iterated composition of the butterfly building block $\mathcal{B}$. This fact is illustrated for $\mathcal{B}_3$ in Fig. 10.

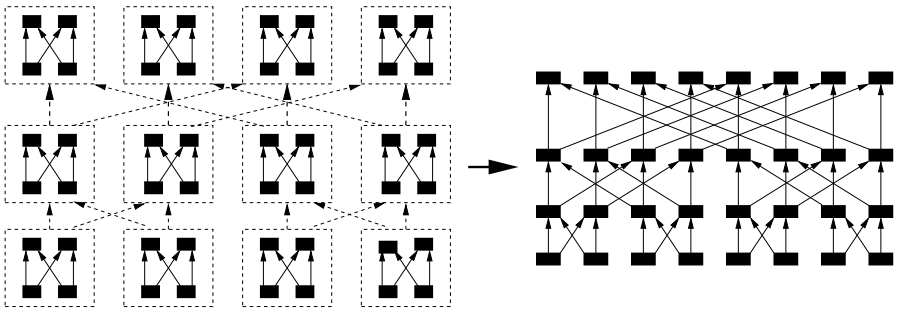

Figure 10: *The butterfly network as a composition of butterfly building blocks.*

A trivial computation using (2.1) shows that $\mathcal{B} \triangleright \mathcal{B}$. It follows that every iterated composition of $\mathcal{B}$—hence, specifically, every butterfly network $\mathcal{B}_d$ is an $\triangleright$-linear composition. By Theorem 2.1, then, every such composition admits an IC-optimal schedule. Indeed, a generalized argument from [23] shows:

> *A schedule for an iterated composition $\mathcal{G}$ of the butterfly building block $\mathcal{B}$ is IC optimal if, and only if, it executes the two sources of each copy of $\mathcal{B}$ within $\mathcal{G}$ in consecutive steps.*

Rendering butterfly-structured computations multi-granular. The literature discusses many computations whose dependency structure is modeled by butterfly networks; cf. [18, 25]. Most such discussions—including ours in Section 5.2—focus on computations having fine-grained tasks. These discussions notwithstanding, butterfly networks support important computations having tasks of arbitrary granularities. This is because every $(a+b)$-dimensional butterfly network $\mathcal{B}_{a+b}$ is (isomorphic to) a copy of $\mathcal{B}_a$ each of whose nodes is a copy of $\mathcal{B}_b$; cf. [1]. Fig. 10 exemplifies this fact for the case $a = 2$ and $b = 1$. This fact allows one both to adjust the granularities of the tasks that are allocated to remote clients and to control the volume of internode communication, while always retaining butterly-structured dependencies.

5.2 Sample Computations

We noted earlier that many transformations effected by butterfly building blocks lead to useful computations. We exemplify just two now: the *comparator transformation*

$$y_0 \;=\; \min(x_0,\, x_1) \quad \text{and} \quad y_1 \;=\; \max(x_0,\, x_1) \tag{5.1}$$

and the *convolution transformation*

$$y_0 \;=\; x_0 \;+\; \omega x_1 \quad \text{and} \quad y_1 \;=\; x_0 \;-\; \omega x_1 \tag{5.2}$$

where ω is a constant associated with this specific butterfly building block. We now describe the complex computations that these transformations lead to.

Sorting. It has been known for decades [2] that some iterated compositions of the butterfly building block—using the comparator transformation (5.1)—will sort any sequence of keys from an ordered domain, that are presented at the sources of the composite dag. In fact, one can derive one such family of networks (though not the most efficient one) via iterated composition of butterfly networks; the most efficient known such networks require a more complicated iterated composition of comparators [11].

In summation, there are sorting algorithms—any comparator-based one will work—that can be computed IC optimally by a simply specified algorithm.

Convolutions. Given two univariate polynomials of degree n,

$$f(x) \;=\; a_0 + a_1x + a_2x^2 + \cdots + a_nx^n \quad \text{and} \quad g(x) \;=\; b_0 + b_1x + b_2x^2 + \cdots + b_nx^n$$

their product is the polynomial

$$[f \otimes g](x) \;=\; A_0 + A_1x + A_2x^2 + \cdots + A_{2n}x^{2n}$$

where each coefficient A_k is a *convolution*, i.e., a sum of the form

$$A_k \;=\; a_0b_k + a_1b_{k-1} + \cdots + a_{k-1}b_1 + a_kb_0 \;=\; \sum_{i=0}^{k} a_ib_{k-i}.$$

Convolutions arise in myriad computations other than polynomial multiplication, one of the most important being the Fast Fourier Transform (FFT); cf. [11]. In fact, the data dependencies of the d-dimensional FFT computation have the form of the butterfly network $\mathcal{B}_d$, hence can be computed IC optimally by a simply specified algorithm. Specifically, each butterfly building block used to construct the FFT computation uses the convolutional transformation (5.2) with a value of ω that is derived from the dth complex roots of unity.

In fact, one can use the FFT computation to perform a large repertoire of convolutions, notably including polynomial multiplication, in sequential time $\Theta(n \log n)$. We thereby can schedule a broad range of convolutional computations IC optimally.

6 A More Complex Expansion-Reduction Paradigm

6.1 The Parallel-Prefix/Scan Operator

The operator and an associated dag. We now describe the *parallel-prefix* (or, *scan*) operator, a meta-computation that provides myriad examples of important computations

that our theory can schedule IC optimally. It has been shown in, e.g., [3, 18], that the ability to compute parallel-prefixes efficiently automatically enables one to compute a large variety of computations efficiently, ranging from microscopic ones such as carry-lookahead addition, to large ones such as we exemplify in Section 6.2.1.

The parallel-prefix operator is defined for an arbitraryy binary associative operation that we denote $*$ (think, e.g., of $+$, $\times$, min, max, "concatenate"). The $*$-parallel prefix of the input vector $\langle x_1, x_2, \ldots, x_n \rangle$ is the output vector $\langle y_1, y_2, \ldots, y_n \rangle$, where

$$\begin{array}{lcll} y_1 & = & x_1 & \\ y_2 & = & y_1 * x_2 & = \quad x_1 * x_2 \\ \vdots & & \vdots & \quad \vdots \\ y_n & = & y_{n-1} * x_n & = \quad x_1 * x_2 * \cdots * x_{n-1} * x_n \end{array} \tag{6.3}$$

There are many ways of implementing the $*$-parallel prefix computation. The following way is attractive because it performs the computation in $O(\log n)$ parallel steps (in the presence of n-fold parallel computation).

for $j = 0$ **to** $\lfloor \log_2(n-1) \rfloor$ **do**
  for $i = 2^j$ **to** $n-1$ **do in parallel**
    $x_i \leftarrow x_{i-2^j} * x_i$
do in parallel $y_i \leftarrow x_i$

The 8-input parallel-prefix dag $\mathcal{P}_8$ that represents this algorithm is depicted in Fig. 11.

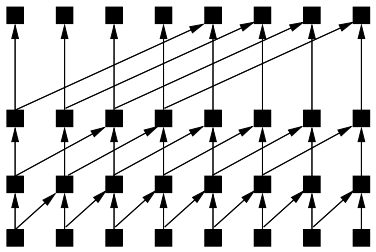

Figure 11: *The* 8*-input parallel-prefix dag* $\mathcal{P}_8$.

IC-optimal schedules for parallel-prefix dags. The easiest way to begin deriving an IC-optimal schedule for the n-input parallel-prefix dag $\mathcal{P}_n$ is to note that it is an iterated composition of *N-dags*.

> For each integer $s > 0$, the s-source *N-dag* $\mathcal{N}_s$ has s sources and s sinks; its $2s - 1$ arcs connect each source v to sink v and sink $v + 1$ if the latter exists. $\mathcal{N}_s$'s leftmost source—the dag's *anchor*—has a child that has no other parents.

This fact is illustrated in Fig. 12, where it is shown that $\mathcal{P}_8$ is composite of type $\mathcal{N}_8 \Uparrow \mathcal{N}_4 \Uparrow \mathcal{N}_4 \Uparrow \mathcal{N}_2 \Uparrow \mathcal{N}_2 \Uparrow \mathcal{N}_2 \Uparrow \mathcal{N}_2$. (The fact that the constituent N-dags shrink is a coincidence; it is not needed for our analysis.) It is not hard to verify—cf. [21]—that: (a) the schedule

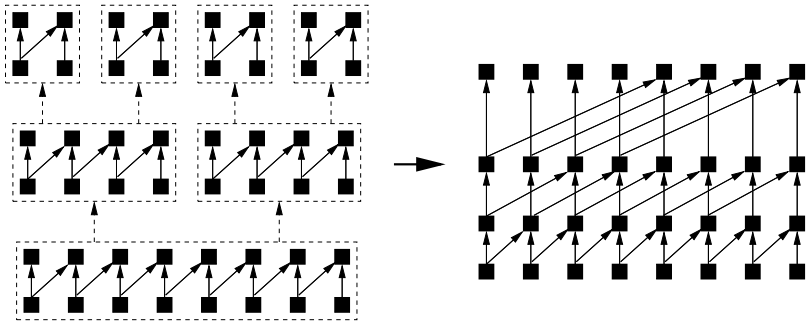

Figure 12: *Exemplifying parallel-prefix dags as compositions of N-dags.*

that executes the sources of $\mathcal{N}_s$ sequentially, starting with the anchor, is IC optimal; (b) $\mathcal{N}_s \rhd \mathcal{N}_t$ for all s and t. These facts combine to show that every dag $\mathcal{P}_n$ is a $\rhd$-linear composition, hence admits an IC-optimal schedule (Theorem 2.1). Indeed:

> *Any schedule that executes $\mathcal{P}_n$ by executing its constituent N-dags in nonincreasing order of their numbers of sources is IC optimal.*

Rendering parallel-prefix computations multi-granular. Since system (6.3) demands only that the "scanned" operation $*$ be associative, one can employ the $*$-parallel-prefix dag $\mathcal{P}_n^{(*)}$ to apply the operator to a rather wide range of operations. For instance:

- to generate the first n powers of an integer N, one lets $*$ be *integer multiplication* and uses input $\langle N, \ldots, N \rangle$;
- to generate the first n powers of an complex number ω, one lets $*$ be *complex multiplication* (which is significantly harder than integer multiplication) and uses input $\langle \omega, \ldots, \omega \rangle$;
- to generate the first n "logical" powers of the adjacency matrix A of an n-node graph $\mathcal{G}$, with the end of computing all paths in $\mathcal{G}$, one lets $*$ be *logical matrix multiplication* (which replaces the arithmetic sum and product of ordinary matrix multiplication by the logical sum (OR) and product (AND)) and uses input $\langle A \ldots, A \rangle$. This, of course, is a considerably more complex operation than multiplication of numbers.

These examples illustrate how one can use the parallel-prefix operator for tasks of varying complexity and coarseness. Of course, to get finer-grain tasks, one can, e.g., expand the operation of matrix multiplication to its constituent scalar operations; cf. Section 7. Of course, we can also modify the granularity of tasks by manipulating the dependency dag, as illustrated in earlier sections.

6.2 Two Sample Computations

6.2.1 The Discrete Laplace Transform

The n-dimensional *Discrete Laplace Transform* (*DLT*, for short)—a/k/a the *Z-Transform*—transforms an n-dimensional vector $\langle x_0, x_1, \ldots, x_{n-1}\rangle$ to an m-dimensional vector of complex functions $\langle y_0(\omega), y_1(\omega), \ldots, y_{m-1}(\omega)\rangle$. The value $y_k(\omega)$ is given (cf. [4]) by

$$y_k(\omega) \;=\; x_0 + x_1\omega^k + x_2\omega^{2k} + \cdots + x_{n-1}\omega^{(n-1)k} \;=\; \sum_{i=0}^{n-1} x_i\omega^{ik}. \tag{6.4}$$

We consider two algorithms for computing the DLT. Both use an in-tree to accumulate the terms of the sum (6.4), but they generate the terms quite differently. We expect each algorithm to be preferable to the other on some platforms—but at least, the two algorithms illustrate quite different dag structures that both admit IC optimal schedules. The in-tree used by both algorithms has n sources. Each source v_i begins by multiplying x_i times the power of ω that v_i has received. Then, in terms of the depiction of Λ in Fig. 1, the variables y_0, y_1, z represent subsums of (6.4):

$$\begin{array}{lrcl} \textbf{if} & y_0 & = & x_i \cdot \omega^{ik} \\ \textbf{and} & y_1 & = & x_j \cdot \omega^{jk} \\ \textbf{then} & z & = & y_0 + y_1 \end{array}$$

We now present the two generating algorithms.

Generate terms via the parallel-prefix operator. For any complex number ω, one can compute $y_k(\omega)$ by using an n-input parallel-prefix dag $\mathcal{P}_n$ with input vector $\langle \omega^k, \omega^k, \ldots, \omega^k\rangle$ to generate the vector $\langle 1, \omega^k, \omega^{2k}, \ldots, \omega^{(n-1)k}\rangle$ as input to the accumulating in-tree. The resulting 8-input composite DLT dag, $\mathcal{L}_8$, is depicted on the lefthand side of Fig. 13.

We simplify the argument that every $\mathcal{L}_n$ admits an IC-optimal schedule by assuming that $n = 2^p$ is a power of 2. Focus on the building blocks of $\mathcal{L}_n$, and note: (a) $\mathcal{L}_n$ is composite of type $\mathcal{P}_n \Uparrow \mathcal{T}_n$, where $\mathcal{T}_n$ is the n-source in-tree; (b) $\mathcal{P}_n$ is composite of type[12]

$$\mathcal{N}_{2^p} \Uparrow (\mathcal{N}_{2^{p-1}} \Uparrow \mathcal{N}_{2^{p-1}}) \Uparrow (\mathcal{N}_{2^{p-2}} \Uparrow \mathcal{N}_{2^{p-2}} \Uparrow \mathcal{N}_{2^{p-2}} \Uparrow \mathcal{N}_{2^{p-2}}) \Uparrow \cdots \Uparrow (\mathsf{N}_2 \Uparrow \mathcal{N}_2 \Uparrow \cdots \Uparrow \mathcal{N}_2)$$

(2^{p-1} copies of "$\mathcal{N}_2$"); (c) $\mathcal{T}_n$ is composite of type $\Lambda \Uparrow \cdots \Uparrow \Lambda$ ($2^p - 1$ copies of "Λ").

The three facts (from [21]) that: (1) $(\forall s, t)$ $[\mathcal{N}_s \rhd \mathcal{N}_t]$; (2) $(\forall s)$ $[\mathcal{N}_s \rhd \Lambda]$; (3) $[\Lambda \rhd \Lambda]$, imply that every $\mathcal{L}_n$ is a $\rhd$-linear composition, hence admits an IC-optimal schedule (Theorem 2.1). Indeed, the theorem indicates that

[12] We add parentheses to the type expression to enhance legibility.

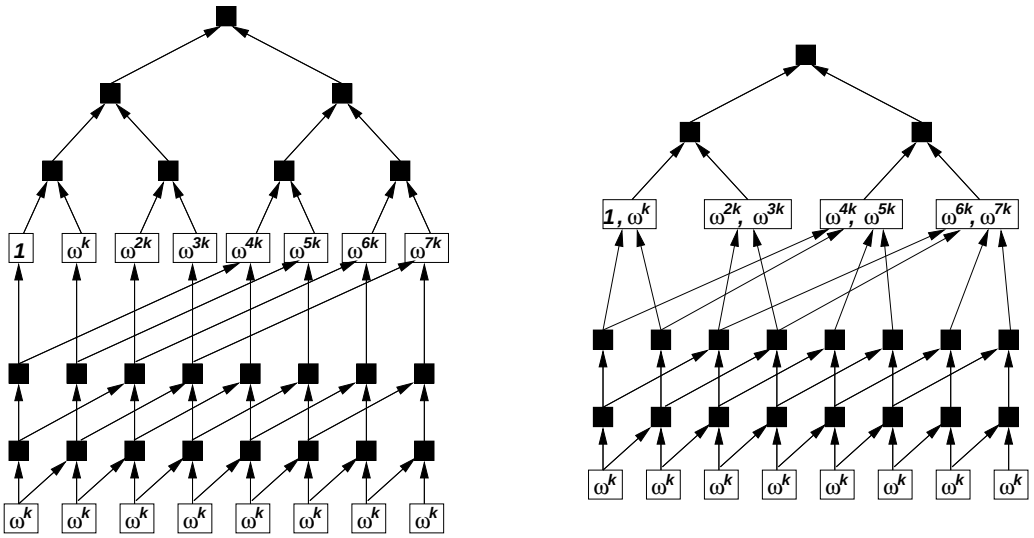

Figure 13: (left) *The* 8*-input DLT dag* $\mathcal{L}_8$*;* (right) *a coarsened version of* $\mathcal{L}_8$*.*

Any schedule that executes $\mathcal{L}_n$ *by executing its constituent copy of* $\mathcal{P}_n$ *IC optimally, then executing its constituent copy of* $\mathcal{T}_n$ *IC optimally, is IC optimal.*

The argument showing that the coarsened version of $\mathcal{L}_8$ depicted on the righthand side of Fig. 13 admits an IC-optimal schedule combines the preceding $\triangleright$-priority-related reasoning with the purely topological fact that the righthand portion of the in-tree cannot be executed until its sources have been executed. Details are left to the reader.

Generate terms via a specialized out-tree. An alternative algorithm for generating the terms of the sum (6.4) employs a *ternary* out-tree that is built out of the 3*-prong Vee dag* $\mathcal{V}_3$ depicted in Fig. 14. In terms of the depiction of $\mathcal{V}_3$ in Fig. 14, the variables

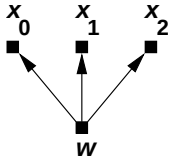

Figure 14: *The* 3*-prong Vee dag* $\mathcal{V}_3$*.*

w, x_0, x_1, x_2 represent powers of the parameter ω^k in (6.4):

$$\begin{array}{llcll} \textbf{if} & w & = & \omega^{ik} & \\ \textbf{then} & x_0 & = & \omega \cdot w^2 & = \omega^{(2i+1)k} \\ \textbf{and} & x_1 & = & w & = \omega^{ik} \\ \textbf{and} & x_2 & = & w^2 & = \omega^{2ik} \end{array}$$

The initial power—the root of the out-tree—represents the parameter ω^k. The 8-input resulting composite dag for the DLT, $\mathcal{L}'_8$, is depicted in Fig. 15. Having presented several

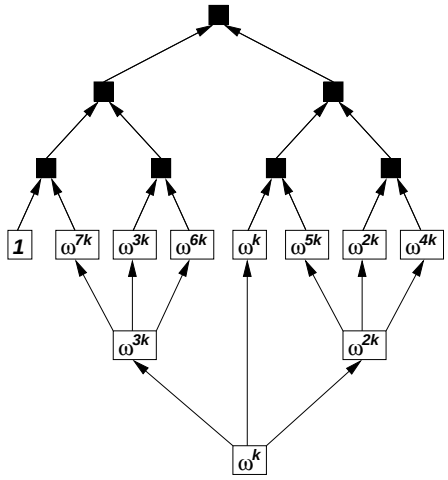

Figure 15: *The alternative* 8*-input DLT dag* $\mathcal{L}'_8$.

such arguments by now, we indicate only sketchily why every $\mathcal{L}'_n$ admits an IC-optimal schedule. One validates easily the chain

$$\mathcal{V}_3 \ \triangleright \ \mathcal{V}_3 \ \triangleright \ \Lambda \ \triangleright \ \Lambda,$$

Thus, every $\mathcal{L}'_n$ is a $\triangleright$-linear composition, hence admits the IC-optimal schedule that executes the out-tree, then the leftmost source, then the in-tree.

6.2.2 Computing paths in a graph

We now consider an operation that, while not familiar, is quite natural. It exemplifies a coarse computation that falls within the framework of this section. We anticipated this computation in our discussion of the multi-granular nature of the parallel-prefix operator in Section 6.1. Consider Fig. 16 as we describe the computation. Say that we have a 9-node graph $\mathcal{G}$, presented via its 9×9 boolean adjacency matrix A. (We choose the integer 9 to make Fig. 16 attractive.) We wish to compute a 9×9 matrix M of integers whose (i, j) entry is a vector $\vec{v}_{i,j} = \langle \beta_{i,j}^{(1)}, \ldots, \beta_{i,j}^{(8)} \rangle$, where

$$\beta_{i,j}^{(k)} \ = \ \begin{cases} 1 & \text{if there is a path of length } k \text{ in } \mathcal{G} \text{ between nodes } i \text{ and } j \\ 0 & \text{if there is no such path} \end{cases}$$

We compute M via a computation whose intertask dependencies are depicted in Fig. 16.

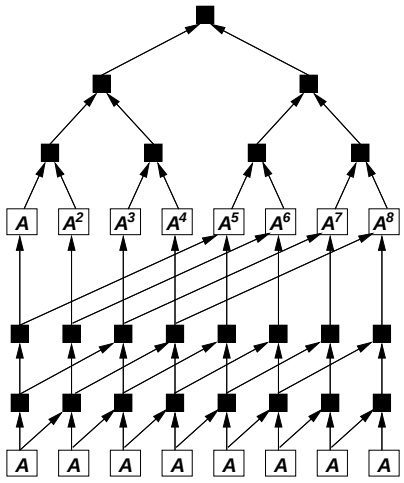

Figure 16: *Computing the paths in a* 9*-node graph.*

1. We use an 8-input parallel-prefix to compute all logical powers of A (as described in Section 6.1). Within each power A^k, the (i, j) entry is 1 or 0, indicating whether or not there is a path of length k in $\mathcal{G}$ between nodes i and j.

2. We use an in-tree to accumulate the information from the eight power matrices A^k into the 64 vectors $\vec{v}_{i,j}$ of matrix M.

There are, of course, many variations on the indicated theme, all of which yield to computations having the structure depicted in Fig. 16.

7 Matrix Multiplication

Our final sample computation is the ubiquitous operation of matrix multiplication. The well-known algorithm that multiplies one $n \times n$ matrix by another via recursive invocation of the 2×2 algorithm (cf. [11]) allows one considerable control over the granularities of tasks. Since we have one specific application in mind here, we invert the order of presentation from other sections.

7.1 Multipying 2×2 Matrices Recursively

The product of the 2×2 matrices $\mathcal{A} = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$ and $\mathcal{B} = \begin{pmatrix} E & F \\ G & H \end{pmatrix}$ is given by:

$$\mathcal{A} \times \mathcal{B} \;=\; \begin{pmatrix} A & B \\ C & D \end{pmatrix} \times \begin{pmatrix} E & F \\ G & H \end{pmatrix} \;=\; \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + BH \end{pmatrix} \tag{7.1}$$

Importantly, the rightmost expression in (7.1) does not invoke the commutativity of multiplication, so the equation holds when the elements of $\mathcal{A}$ and $\mathcal{B}$ are themselves matrices. Thus, (7.1) actually specifies a recursive algorithm for multiplying any $n \times n$ matrix by another. Each level of this recursion has the simple structure exposed in Fig. 17.

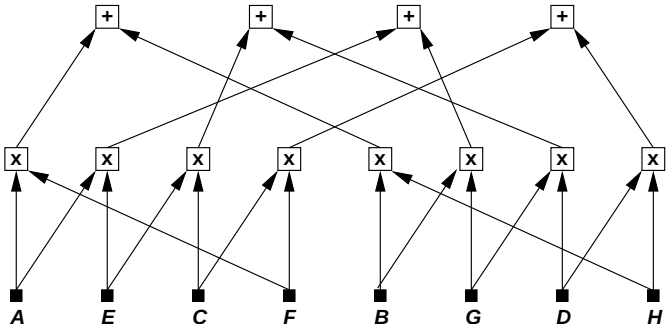

Figure 17: *The dag $\mathcal{M}$: Multiplying 2×2 matrices, or, via recursion, $n \times n$ matrices.*

7.2 IC-Optimal Schedules for Our Matrix-Multiplication Dag

The structure of the dag in Fig. 17 can be elucidated in terms of *(bipartite) cycle-dags*.

> For each integer $s > 1$, the s-source *(bipartite) cycle-dag* $\mathcal{C}_s$ is obtained from the N-dag $\mathcal{N}_s$ by adding a new arc from the rightmost source to the leftmost sink—so that each source v of $\mathcal{C}_s$ has arcs to sinks v and $v + 1 \bmod s$.

One notes in Fig. 17 that our matrix-multiplication dag $\mathcal{M}$ contains two copies of $\mathcal{C}_4$, one to compute the products AE, AF, CE, CF and one for BG, BH, DG, DH. These copies are composed with four copies of Λ that compute the required sums of these products. In fact, $\mathcal{M}$ is composite of type $\mathcal{C}_4 \Uparrow \mathcal{C}_4 \Uparrow \Lambda \Uparrow \Lambda \Uparrow \Lambda \Uparrow \Lambda$. A simple calculation using (2.1) verifies that $\mathcal{C}_4 \rhd \mathcal{C}_4 \rhd \Lambda \rhd \Lambda$ (cf. [21]). Thus, $\mathcal{M}$ is a $\rhd$-linear composition, hence admits an IC-optimal schedule (Theorem 2.1), one of which is specified as follows.

> *The following is an IC-optimal schedule for multiplying 2×2 matrices. Compute the eight required products in the order $AE, CE, CF, AF, BG, DG, DH, BH$. Then compute the four required sums involving these products, in any order.*

As with previous sample computations, refining or clustering the dependency dag allows one to adjust the granularity of the computation's constituent tasks.

8 Where We Are, and Where We're Going

In Sections 5–7, we present a broad range of computational structures that yield to IC-scheduling theory and that arise in significant "real" computations. We have exploited results only from [21, 9], which develop the existing algorithmics of IC-scheduling theory. We expect the upcoming sequel [10] to these sources to enable the IC-optimal scheduling of additional computational structures, which arise in other significant "real" computations.

Our current research priorities focus on four broad thrusts for extending and/or refining IC-scheduling theory:

1. enabling the optimal scheduling of ever broader classes of dags;
2. developing rigorous notions of "almost" optimal scheduling that apply to *all* dags (which is important since the strong demands of IC optimality preclude the IC-optimal scheduling of many dags [21]);
3. incorporating concerns such as communication load, which are critically important to IC;
4. extending the assessment component of IC-scheduling theory, both via further simulation experiments, as in [15, 19], and via experimentation using actual IC systems, as enabled by the PRIO tool of [19] and its potential successors.

Acknowledgments. The research of A. Rosenberg was supported in part by NSF Grant CCF-0342417.

References

[1] A. Avior, T. Calamoneri, S. Even, A. Litman, A.L. Rosenberg (1998): A tight layout of the butterfly network. *Theory of Computing Systs. 31* (Special issue for *SPAA/96*) 475–487.

[2] V.E. Beneš (1964): Optimal rearrangeable multistage connecting networks. *Bell Syst. Tech. J. 43*, 1641–1656.

[3] G.E. Blelloch (1989): Scans as primitive parallel operations. *IEEE Trans. Comput. 38*, 1526–1538.

[4] L.I. Bluestein (1970): A linear filtering approach to the computation of the Discrete Fourier Transform. *IEEE Trans. Audio Electroacoust., AU-18*, 451–455.

[5] R. Buyya, D. Abramson, J. Giddy (2001): A case for economy Grid architecture for service oriented Grid computing. *10th Heterogeneous Computing Wkshp.*

[6] W. Cirne and K. Marzullo (1999): The Computational Co-Op: gathering clusters into a metacomputer. *13th Intl. Parallel Processing Symp.*, 160–166.

[7] Condor Project, University of Wisconsin. `http://www.cs.wisc.edu/condor`

[8] S.A. Cook (1974): An observation on time-storage tradeoff. *J. Comp. Syst. Scis. 9*, 308–316.

[9] G. Cordasco, G. Malewicz, A.L. Rosenberg (2006): Advances in a dag-scheduling theory for Internet-based computing. Submitted for publication. See also, On scheduling expansive and reductive dags for Internet-based computing. *26th Intl. Conf. on Distributed Computing Systems*, 2006.

[10] G. Cordasco, G. Malewicz, A.L. Rosenberg (2006): Extending IC-Scheduling Theory via the Sweep Algorithm. Typescript, Univ. Massachusetts.

[11] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein (1999): *Introduction to Algorithms* (2nd Edition). MIT Press, Cambridge, Mass.

[12] I. Foster and C. Kesselman [eds.] (2004): *The Grid: Blueprint for a New Computing Infrastructure (2nd Edition)*. Morgan-Kaufmann, San Francisco.

[13] I. Foster, C. Kesselman, S. Tuecke (2001): The anatomy of the Grid: enabling scalable virtual organizations. *Intl. J. Supercomputer Applications*.

[14] L. Gao and G. Malewicz (2006): Toward maximizing the quality of results of dependent tasks computed unreliably. *Theory of Computing Systs.*, to appear. See also, *Intl. Conf. on Principles of Distributed Systems*, 2004.

[15] R. Hall, A.L. Rosenberg, A. Venkataramani (2006): A comparison of dag-scheduling strategies for Internet-based computing. Typescript, Univ. Massachusetts.

[16] D. Kondo, H. Casanova, E. Wing, F. Berman (2002): Models and scheduling mechanisms for global computing applications. *Intl. Parallel and Distr. Processing Symp.*

[17] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, M. Lebofsky (2000): SETI@home: massively distributed computing for SETI. In *Computing in Science and Engineering* (P.F. Dubois, Ed.) IEEE Computer Soc. Press, Los Alamitos, CA.

[18] F.T. Leighton (1992): *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, San Mateo, Cal.

[19] G. Malewicz, I. Foster, A.L. Rosenberg, M. Wilde (2006): A tool for prioritizing DAGMan jobs and its evaluation." *15th IEEE Intl. Symp. on High-Performance Distributed Computing*, 156–167.

[20] G. Malewicz and A.L. Rosenberg (2005): On batch-scheduling dags for Internet-based computing. *Euro-Par 2005*. In *Lecture Notes in Computer Science 3648*, Springer-Verlag, Berlin, 262–271.

[21] G. Malewicz, A.L. Rosenberg, M. Yurkewych (2006): Toward a theory for scheduling dags in Internet-based computing. *IEEE Trans. Comput. 55*, 757–768.

[22] A.L. Rosenberg (2004): On scheduling mesh-structured computations for Internet-based computing. *IEEE Trans. Comput. 53*, 1176–1186.

[23] A.L. Rosenberg and M. Yurkewych (2005): Guidelines for scheduling some common computation-dags for Internet-based computing. *IEEE Trans. Comput. 54*, 428–438.

[24] X.-H. Sun and M. Wu (2003): GHS: A performance prediction and node scheduling system for Grid computing. *IEEE Intl. Parallel and Distributed Processing Symp.*

[25] J.D. Ullman (1984): *Computational Aspects of VLSI*. Computer Science Press, Rockville, Md.